Optimization of Tree-Decomposition-based Dynamic Programming for AI Problems*

Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran

Institute of Information Systems 184/2
TU Wien
Favoritenstrasse 9–11, 1040 Vienna, Austria
[bliem,gcharwat,hecher,woltran]@dbai.tuwien.ac.at

Abstract

Many problems from the area of AI have been shown tractable for bounded treewidth. In order to put such results into practice, quite involved dynamic programming (DP) algorithms on tree decompositions have to be designed and implemented. These algorithms typically show recurring patterns that call for tasks like subset-minimization. In this paper, we provide a new method for obtaining DP algorithms from simpler principles, where subset-minimization is performed automatically. Moreover, we put this method into practice by implementing DP algorithms for AI problems in a more space-efficient way than existing solutions. Experiments show that our approach also yields a significant improvement in runtime performance.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases tree decomposition, dynamic programming, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Many prominent NP-hard problems in the area of AI have been shown tractable for bounded treewidth, see, e.g., [14, 16, 9]. Thanks to Courcelle's meta-theorem [4], it is sufficient to encode a problem as an MSO sentence in order to obtain such a result. To put this into practice, tailored systems for MSO logic are required, however. While there has been remarkable progress in this direction [20] there is still evidence that designing DP algorithms for the considered problems from scratch results in more efficient software solutions (cf. [23]).

The actual design of these algorithms can be quite tedious, in particular for problems located at the second level of the polynomial hierarchy like circumscription, abduction, answer set programming or abstract argumentation (see [10, 18, 17, 15]). In many cases, the increased complexity of such problems is caused by subset minimization or maximization subproblems (e.g., minimality of models in circumscription). It is exactly the handling of these subproblems which makes the design of the DP algorithms difficult.

Especially in the world of answer set progamming (ASP), we can witness such a phenomenon: In order to exploit the full expressive power of this paradigm, a particular saturation programming technique is required (see [21]) in order to express co-NP tests. Several approaches for relieving the user from this task have been proposed [12, 13]. For instance, in order to find minimal propositional models in ASP, we simply need to express the SAT problem together with a special minimize statement (recognized by systems like

* This work was supported by the Austrian Science Fund (FWF) projects P25607 and Y698.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

metasp). In this way, we easily obtain a program computing minimal models. Unfortunately, easy-to-use facilities like such minimize statements had no analog in the area of DP so far.

In this paper, we propose a solution to this issue: We provide a method for automatically obtaining DP algorithms for problems requiring minimization, given only an algorithm for a problem variant without minimization. For example, given a DP algorithm for Sat (like in [26]), our approach makes it possible to use this algorithm, together with simple statements on what to minimize, for finding only subset-minimal models. Making minimization implicit in this way makes the programmer's life considerably easier.

Furthermore, standard DP algorithms on such problems often suffer from a naive check for subset minimality that requires unnecessarily much space and time. Our main contribution is providing a method that avoids this issue by proceeding in two stages (instead of one stage in the naive case): First we compute solution candidates without regard to minimization and then we rule out invalid candidates by trying to find counterexamples. Because of its two-phased nature, our approach can do so in an efficient way. We are not aware of any work so far that introduces similar two-phased DP algorithms along with the appropriate data structures. To underline the practical relevance of our work, we present promising experimental results on AI problems. Although we explicitly only consider minimization, all the results in this paper of course also apply to maximization.

2 Background

In this section we outline dynamic programming on tree decompositions. The ideas underlying this concept stem from the field of parameterized complexity. Many computationally hard problems become tractable in case a certain problem parameter is bound to a fixed constant. This property is referred to as *fixed-parameter tractability* (fpt) [7], and the complexity class FPT consists of problems that are solvable in $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a function that only depends on the parameter k, and n is the input size.

Tree decompositions. For problems whose input can be represented as a graph, one important parameter is *treewidth*, which, roughly speaking, measures the "tree-likeness" of a graph. It is defined by means of tree decompositions (TDs), originally introduced in [25]. The intuition behind TDs is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices under one node and thereby isolating the parts responsible for cyclicity.

▶ **Definition 1.** A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's *bag*), such that the following conditions are met: **1.** For every vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$. **2.** For every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$. **3.** For every $v \in V$, the subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected.

We call $\max_{n \in N} |\chi(n)| - 1$ the *width* of the decomposition. The *treewidth* of a graph is the minimum width over all its tree decompositions.

In general, constructing a TD with minimum width is intractable [2]. However, there are heuristics that give "good" TDs in polynomial time [5, 6, 3]. In this paper we will consider so-called *semi-normalized* TDs:

▶ **Definition 2.** A tree decomposition $\mathcal{T} = (T, \chi)$ with $T = (N, F)$ is *semi-normalized* if each non-leaf node $n \in N$ is an *exchange node* (n has exactly one child), or a *join node* (n has exactly two children n', n'' with $\chi(n) = \chi(n') = \chi(n'')$).

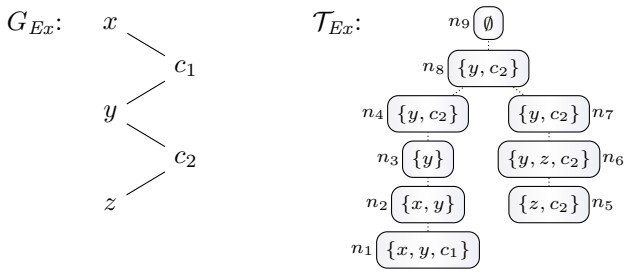

Figure 1 Incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$ of ϕ_{Ex}.

Furthermore, we assume that for root node r of T, $\chi(r) = \emptyset$ holds. A TD can be transformed into a semi-normalized one in linear time without increasing the width [19].

The enumeration variants of the Sat and $\subseteq$-Minimal Sat problems ("Given a propositional formula ϕ in CNF, what are the (subset-minimal) models of ϕ?") will serve as running examples throughout this section. These problems are well suited since the DP algorithms incorporate concepts that often reappear in other AI-related problem domains. To obtain a TD of a CNF-formula ϕ, we first have to construct an appropriate graph representation. Let $\mathcal{C}$ denote the set of clauses and $\mathcal{A}$ the atoms in ϕ. Furthermore, let $at(c)$ denote the atoms occurring in $c \in \mathcal{C}$. Then, the *incidence graph* $G = (V, E)$ of ϕ is given as $V = \mathcal{C} \cup \mathcal{A}$ and $E = \bigcup_{c \in \mathcal{C}} \{\{a, c\} \mid a \in at(c)\}$.

▶ **Example 3.** Let $\phi_{Ex} = (x \vee y) \wedge (\neg y \vee z)$. We have $\mathcal{C} = \{c_1, c_2\}$ with $at(c_1) = \{x, y\}$ and $at(c_2) = \{y, z\}$. The corresponding incidence graph G_{Ex} and a possible TD $\mathcal{T}_{Ex}$ are depicted in Figure 1. The width of $\mathcal{T}_{Ex}$ is 2.

Dynamic programming on tree decompositions. Algorithms for DP on TDs generally traverse the TD in bottom-up order. At each node, partial solutions for the subgraph induced by the vertices encountered so far are computed and stored in a data structure associated with the node. The size of the data structure is typically bounded by the width of the TD and the number of TD nodes is linear in the input size. Hence, if the width is bounded by a constant, the search space for the subproblem is constant as well, and the number of subproblems only grows by a linear factor for larger instances.

In the following, so-called *item trees* [1] will serve as the data structure for storing partial solutions. In Section 3 we will present modifications that allow us to solve several problems more efficiently that are hard for the second level of the polynomial hierarchy. Each item tree node contains an *item set* whose elements are called *items*. Usually, the information stored in the items is restricted to (or dependent on) the bag elements of the respective decomposition node. Each item tree node additionally has a set of *extension pointer tuples* that represents its origin. With this, one can reconstruct solutions for the complete problem instance by starting at the root of the TD and following the extension pointer tuples while combining the contents of the respective item sets. These concepts are formalized as follows:

▶ **Definition 4.** Let $\mathcal{T} = (T, \chi)$ with $T = (N, F)$ be a tree decomposition, and let $n \in N$. The *item tree* of n is a triple (S_n, X_n, Y_n) where $S_n = (T_n, E_n)$ is a rooted tree. Furthermore, X_n is a function that assigns to each *item tree node* $t_n \in T_n$ an *item set*, where each item is some arbitrary string. Let $n_1, \ldots, n_x \in N$ be the child nodes of n in T, let t_n be an item tree node in S_n, and let, for $1 \leq i \leq x$, T_{n_i} be the item tree nodes in S_{n_i}. Then, Y_n is a function that assigns to each t_n a non-empty set of *extension pointer tuples*, where each tuple is of the form $(t_1, \ldots, t_x)$, such that $t_i \in T_{n_i}$ for $1 \leq i \leq x$. Finally, we inductively define the set of *extensions* of t_n as $Z(t_n) = \{X_n(t_n) \cup A \mid A \in \bigcup_{(p_1,\ldots,p_x) \in Y_n(t_n)} \{e_1 \cup \cdots \cup e_x \mid e_i \in Z(p_i)\}\}$.

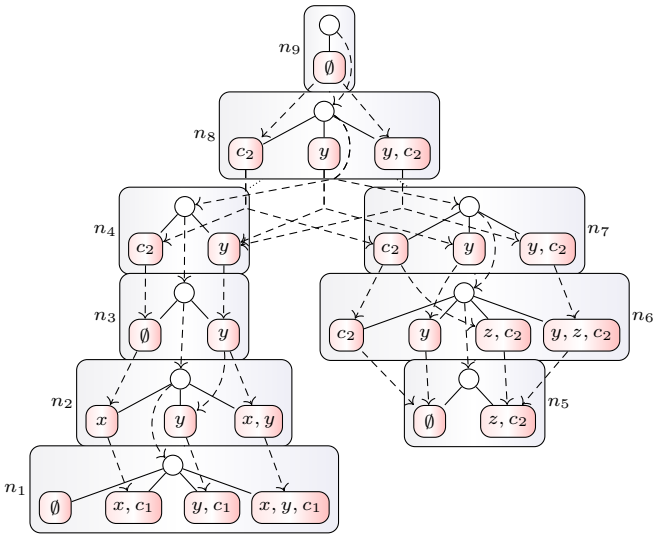

Figure 2 Item trees for SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

For a DP algorithm solving the SAT problem on a TD of the input formula's incidence graph, we store partial solutions in the item sets at depth 1 of the item trees. For a TD node n, these item sets are subsets of $\chi(n)$. Intuitively, each item set represents a partial interpretation for ϕ, together with the clauses satisfied by the interpretation, restricted to $\chi(n)$. At each decomposition node n, we obtain an item tree node by extending one node from the item tree of each child of n. If n is an exchange node with child n', we additionally guess for every introduced atom (i.e., from $\chi(n) \setminus \chi(n')$), whether it is true and if so, we add it to the item set. If a clause from $\chi(n)$ is satisfied by the interpretation represented by this item set, we additionally store that clause in the set. Whenever an atom a is removed from the bag (i.e., in $\chi(n') \setminus \chi(n)$), a is not put into an item set at n. For a clause c that is removed from the bag, we only extend item sets that contained c, as other item sets represent partial interpretations that do not satisfy c. In join nodes, we extend pairs of item tree nodes that coincide on the partial interpretations, and compute the union of the already satisfied clauses. Intuitively, the partial interpretations have to agree on the truth assignment for common atoms in order to be a solution for the complete problem instance.

▶ **Example 5.** Figure 2 illustrates the item trees for $\mathcal{T}_{Ex}$ of ϕ_{Ex}. For instance, in n_1 we store the partial interpretations $\emptyset, \{x\}, \{y\}, \{x, y\}$. The latter three satisfy clause c_1, which is additionally stored in the respective item sets. In n_2, c_1 is removed. Interpretation $\emptyset$ is not extended, since it does not satisfy c_1. On the other hand, the item tree node in n_2 containing $\{x\}$, for example, extends the item tree node $\{x, c_1\}$ in n_1. In the figure, extension pointer tuples are marked with dashed lines. In n_8, the item tree node with item set $\{y, c_2\}$ extends $\{y\}$ in n_4 and $\{y, c_2\}$ in n_7, since the latter two both contain the same partial interpretation. Here, the extension pointer tuple is of arity 2 and contains references to both extended item tree nodes. For enumerating the solutions, we follow the extension pointer tuples, starting at the root of the decomposition, and build the union over the item sets, resulting in $\{\{x, c_1, c_2\}, \{x, z, c_1, c_2\}, \{y, z, c_1, c_2\}, \{x, y, z, c_1, c_2\}\}$. The models of ϕ_{Ex} are $\{\{x\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$.

Next, let us consider the $\subseteq$-MINIMAL SAT problem. Here, solution candidates are again stored at depth 1 of item trees. Furthermore, we store so-called *counter candidates* at depth 2. A counter candidate is a potential witness for the solution candidate (its parent) not being

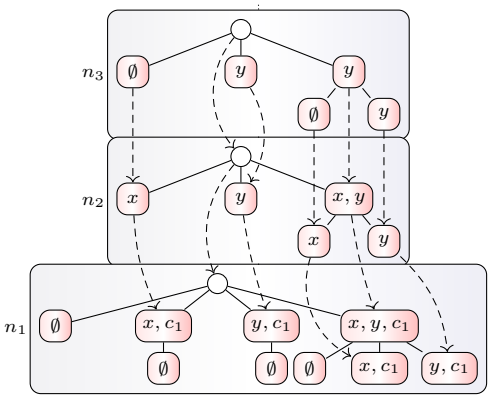

Figure 3 Item trees for $\subseteq$-Minimal Sat of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

subset-minimal. This concept of storing witnesses (also called *certificates*) is commonly used for problems that are hard for the second level of the polynomial hierarchy [18]. Item sets for solution and counter candidates are computed as for the Sat problem. Additionally, partial interpretations represented by counter candidates are strict subsets of partial interpretations represented by solution candidates. In the following, we give details on how and when these counter candidates are constructed. For an introduced atom a in decomposition node n, whenever we add a to the item set of some solution candidate t_n, we also add it to the item sets at the children of t_n. Furthermore, a new child for t_n is added whose item set contains the same atoms and clauses as the extended item set of t_n. This child represents a new potential witness for the partial interpretation associated with t_n being not subset-minimal. In join nodes, solution candidates are extended as in the Sat problem. Furthermore, we construct associated counter candidates by combining counter candidates or a counter candidate of one item tree with the solution candidate of the other item tree, whenever they coincide on the partial interpretations. Intuitively, a counter candidate at the join node either represents a smaller interpretation in both child item trees, or in one of them together with the solution candidate stored in the other item set. For removed atoms, and introduced and removed clauses, the item sets are updated as described for Sat. At the root node, item tree nodes at depth 1 without associated counterexamples represent subset-minimal models of the overall problem instance, and the models are obtained by following the respective extension pointer tuples.

▶ **Example 6.** Consider again $\mathcal{T}_{Ex}$ of ϕ_{Ex}. Figure 3 contains the computed item trees for TD nodes n_1, n_2 and n_3. In n_1, we construct the item sets at depth 1 as described before. The item sets at depth 2 represent counter candidates that are strict subsets of the interpretations at depth 1. In n_2, clause c_1 is removed. Hence, we remove all item sets representing interpretations that do not satisfy c_1. In n_3, atom x is removed. Observe that this results in two item sets at depth 1 that both solely contain y. However, they differ in the counter candidates stored at depth 2.

3 Optimizing Dynamic Programming on TDs

We give a general algorithm of how the task of subset minimization in DP on TDs can be implemented efficiently. Our algorithm is motivated by the following observations. 1) Counter candidates are constructed similarly to solution candidates. 2) Typically, counter candidates are also stored as solution candidates. 3) A counter candidate may turn out to be no solution

Input: Item tree rooted at a node r

```
1  foreach tuple ∈ r.extPtrs, e ∈ tuple do
2  |  computeLv2(e);
3  foreach c ∈ r.children do
4  |  if r belongs to a leaf node then
5  |  |  c.counterC ← c.counterC ∪ {(c, ⊥)};
6  |  else
7  |  |  foreach tuple ∈ c.extPtrs do
8  |  |  |  if r belongs to an exchange node then
9  |  |  |  |  cCs ← handleExchange(c, tuple);
10 |  |  |  else
11 |  |  |  |  cCs ← handleJoin(c, tuple);
12 |  |  |  insertCompress(c, tuple, cCs);
```

Algorithm 1: The procedure `computeLv2`.

after all (e.g., it is no model). Our approach avoids redundant computations of solution and counter candidates. It supports minimization on user-specified items (e.g., on atoms, but not on clauses, for $\subseteq$-Minimal Sat).

In our approach so-called *reduced item trees* serve as the main data structure. A reduced item tree is an item tree of height 1 that can store additional information at each node n: Besides extension pointer tuples, n is associated with a set of *back pointers*. Node n has a back pointer to a node n' iff some element of some extension pointer tuple in n' references n. Furthermore, n has an *optimization item set*, which contains the items on which subset minimization is performed. In contrast to the straightforward way of using (non-reduced) item trees whose depth-2 nodes represent counter candidates, n contains a set of *counter candidate pointers*. A counter candidate pointer is a tuple (c, s), where c is a reference to some sibling of n and s is a flag that is set iff there is an extension of c whose optimization items form a proper subset of those of each extension of n. Note that every node n has at least the counter candidate pointers $(n, \top)$ or $(n, \bot)$.

Opposed to classical implementations such as the algorithm for $\subseteq$-Minimal Sat described in Section 2, we perform two bottom-up traversals of the TD: In the first traversal, we compute all reduced item trees as in classical implementations, but only up to depth 1. After this step, the back pointers are added to the reduced item tree nodes. In the second traversal, we add counter candidates to nodes at depth 1 appropriately, but instead of creating children that are copies of other item sets from depth 1, we store only counter candidate pointers to already existing item sets.

Our main contribution is outlined in Algorithms 1, 2 and 3 (assuming TDs whose leaves have empty bags): The second TD traversal is initiated by applying `computeLv2` to the root of the reduced item tree at the TD's root. This results in all counter candidates being appended to the reduced item trees.

To begin with, we describe the notation employed in our pseudocode. Let n be a reduced item tree node. By n.extPtrs we denote its set of extension pointer tuples, n.children refers to its set of children, n.backPtrs is its set of back pointers, n.items is its set of items, and n.optItems is a set consisting of those items at n among which minimization is performed. Finally, if n is at depth 1 of the reduced item tree, n.counterC is a set of counter candidates of the form (c, s). At the TD's root, these are used to delete any node n at depth 1 of the reduced item tree that has among n.counterC a tuple $(n', \top)$, which witnesses that n' represents a better solution than n.

Input: Item tree node c, extension pointer tuple (e)

13 $cCs \leftarrow \emptyset$;
14 **foreach** $(cc, strict) \in e$.counterC **do**
15 | **foreach** $b \in cc$.backPtrs **do**
16 | | **if** b.optItems $\subseteq c$.optItems **then**
17 | | | $s \leftarrow strict \vee (b.\text{optItems} \subset c.\text{optItems})$;
18 | | | $cCs \leftarrow cCs \cup \{(b, s)\}$;
19 **return** cCs;

Algorithm 2: The function `handleExchange`.

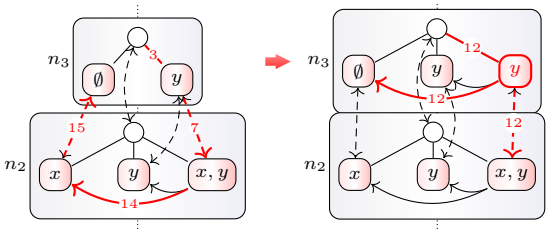

Figure 4 Item tree computation for exchange node n_3 of $\mathcal{T}_{Ex}$.

The omitted procedure `insertCompress`$(c, tuple, cCs)$ sets c.extPtrs to c.extPtrs$\setminus\{tuple\}$, duplicates node c resulting in c' and adds c' to the reduced item tree (as a sibling of c) with c'.counterC set to cCs and c'.extPtrs set to $\{tuple\}$; moreover, c.backPtrs and c'.backPtrs get recomputed. We will discuss the reason for this duplication below (and it will turn out that in some cases we can avoid duplication).

Exchange nodes. We show how Algorithm 2 proceeds at exchange nodes by means of an example illustrated in Figure 4, which consists of two parts: The left hand side depicts two reduced item trees belonging to an exchange node n_3 and its child n_2, respectively, just before the counter candidates at node n_3 are computed. The result of this computation is depicted on the right hand side. Straight solid lines signify parent-child relationships in an reduced item tree; dashed arrows represent extension pointers and back pointers; a curved solid arrow denotes that the target node represents a counter candidate to the source node. (All such arrows depicted in our figures correspond to counter candidates whose subset relation is proper. Note that we omitted self loops via such arrows, which are in fact present at every node because any solution candidate is a counter candidate to itself.) In the following, we explain the figure in detail.

Assuming that all solution candidates have already been obtained, the idea for computing the counter candidates at n_3 is as follows: For each solution candidate C at n_3, we look at the solution candidates at n_2 reachable by extension pointers. For these we already know the counter candidates, as we are doing a bottom-up traversal. We iterate over all of these counter candidates and check if the current one has some back pointer referencing a sibling of C called C' whose set of optimization items is a subset of the respective item set of C. If so, we conclude that C' is a counter candidate to C.

To see an example for such a situation, let $n_{i:S}$ in the following denote the reduced item tree node at n_i whose item set is S in the left hand side of Figure 4. Assume that our invocation of `computeLv2` is currently in a state where the variable c in Line 3 is $n_{3:\{y\}}$ and the variable $tuple$ in Line 7 has the value $(n_{2:\{x,y\}})$. We moreover assume that the tuple of variables $(cc, strict)$ in Line 14 is set to $(n_{2:\{x\}}, \top)$, and the variable b in Line 15 is $n_{3:\emptyset}$. This state of our invocation of `computeLv2` is indicated in the left part of Figure 4 by red

Input: Item tree node c, extension pointer tuple (e_1, e_2)

20 $(cCs, e_1', e_2') \leftarrow (\emptyset, \emptyset, \emptyset)$;
21 **foreach** $e_i \in \{e_1, e_2\}, (cc, strict) \in e_i$.counterC **do**
22 $\quad e_i' \leftarrow e_i' \cup \{cc\}$;
23 **foreach** $(i_1, i_2) \in e_1' \times e_2'$ **do**
24 $\quad$ **foreach** $b \in$ `occurExtPtrs`$((i_1, i_2), c)$ **do**
25 $\quad\quad strict_1 \leftarrow (i_1, \top) \in e_1$.counterC;
26 $\quad\quad strict_2 \leftarrow (i_2, \top) \in e_2$.counterC;
27 $\quad\quad cCs \leftarrow cCs \cup \{(b, strict_1 \vee strict_2)\}$;
28 **return** cCs;

Algorithm 3: The function `handleJoin`.

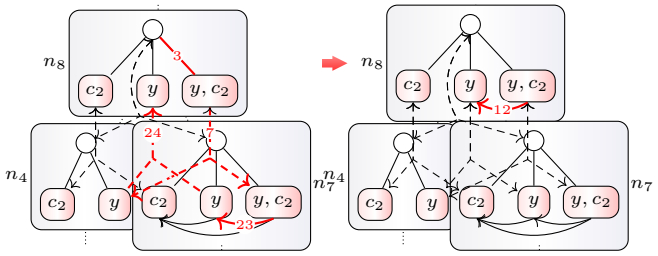

Figure 5 Item tree computation for join node n_8 of $\mathcal{T}_{Ex}$.

arrows whose attached numbers indicate the line number of the corresponding loop. Since $n_{3:\emptyset}$.optItems $\subset n_{3:\{y\}}$.optItems, the set cCs in Line 18 grows by $(n_{3:\emptyset}, \top)$ in the current iteration of the loop. Similarly, the next iteration of the loop in Line 14 causes cCs to grow by $(n_{3:\{y\}}, \top)$ and the current call to `handleExchange` returns $\{(n_{3:\emptyset}, \top), (n_{3:\{y\}}, \top)\}$.

As shown on the right hand side of Figure 4, the subsequent call to `insertCompress` splits node $n_{3:\{y\}}$ into two nodes having the same item set: One of these copies extends $n_{2:\{y\}}$ while the other one extends $n_{2:\{x,y\}}$. In the former case, no counter candidates exist (as $n_{2:\{y\}}$ has none), whereas in the latter we can obtain counter candidates just as the preceding call to `handleExchange` indicated. Splitting nodes like this has the reason that the counter candidates of a solution candidate C depend on which extension pointer of C we choose.

In some cases we can avoid this duplication – in fact, the case of $n_{3:\{y\}}$ is one of them: The rightmost copy of $n_{3:\{y\}}$ on the right hand side of Figure 4 has among its counter candidates a node with the same item set. So if that copy turns out to lead to a solution, then that counter candidate will lead to a proper counterexample: As soon as two nodes have the same item set, they are indistinguishable by DP algorithms.

Join nodes. At join nodes, extension pointer tuples have arity 2, so back pointers are not just inverted extension pointers. This complicates the algorithm compared to the case of exchange nodes. Algorithm 3 shows how we handle join nodes.

To explain the algorithm, we follow an example illustrated in Figure 5. Let $n_{i:S}$ in the following denote the reduced item tree node at n_i whose item set is S in the left hand side of Figure 5. Assume that our invocation of `computeLv2` is currently in a state as depicted in Figure 5, where the variable c in Line 3 is $n_{8:\{y,c_2\}}$ and the variable *tuple* in Line 7 has the value $(n_{4:\{y\}}, n_{7:\{y,c_2\}})$. We moreover assume that the tuple of variables (i_1, i_2) in Line 23 is set to $(n_{4:\{y\}}, n_{7:\{y\}})$; note that in Figure 5 we have omitted the arrow for the first component as it is one of the self-loops that we have hidden. In Line 24, procedure

`occurExtPtrs`$(tuple, c)$ provides a set of nodes such that for each b in the set we have $tuple \in b$.extPtrs and b.optItems $\subseteq c$.optItems. Intuitively, `occurExtPtrs` uses the backward pointers to compute all the extension pointer occurrences, i.e., nodes that have the given 2-tuple among their extension pointer tuples. In our example, for tuple $(n_{4:\{y\}}, n_{7:\{y\}})$, variable b is $n_{8:\{y\}}$. Line 12 inserts the found node $n_{8:\{y\}}$ to the counter candidates of $n_{8:\{y,c_2\}}$, as depicted on the right hand side of Figure 5. The computation of the strict flag in Line 27 is not depicted; it is the disjunction of the flags from the two counter candidates that are extended by the new counter candidate.

Further optimizations. After the counter candidates at a TD node have been computed, all counter candidate pointers at children in the TD can be discarded, which allows for all reduced item tree nodes to be deleted that are no longer referred to by an extension pointer. A further optimization is to remove reduced item tree nodes n with $(n', \top) \in n$.counterC such that n.items $= n'$.items.

As for some problems the counter candidates are not necessarily solution candidates, we propose to mark them with a special flag as "pseudo solution candidates" that only serve to be referenced as counter candidates. To avoid unnecessary memory consumption, such candidates can be deleted as soon as the parent node in the TD has been fully processed.

Example applications. In the propositional case of Circumscription [22] we are given a theory T and sets of atoms P and Z, and we are interested in models M of T such that there is no model M' with $M' \cap P \subset M \cap P$ and $M \cap Z = M' \cap Z$. We can model this in our approach by a slight modification of our $\subseteq$-Minimal Sat algorithm: We only put an atom a in an optimization item set if $a \in P$; and for any $z \in Z$ we add optimization items $t(z)$ or $f(z)$ if the item set contains z or not, respectively (thus making solution candidates with different interpretations of Z incomparable). Given this, we can also easily solve, e.g., disjunctive ASP, as it can be reduced to Circumscription [24].

Further examples where our approach is reasonable are problems from abstract argumentation [8]: Given an object (A, R), where A is a set of arguments and $R \subseteq A \times A$, we call a set $S \subseteq A$ *admissible* if (1) $(a, b) \notin R$ for all $a, b \in S$ and (2) for each $s \in S$ and $r \in A$, $(r, s) \in R$ implies that there is some $q \in S$ with $(q, r) \in R$. S is *preferred* if it is a subset-maximal admissible set. For any $C \subseteq A$, let $C^+ = C \cup \{a \mid \exists b \in C \text{ s.t. } (b, a) \in R\}$. S is *semi-stable* if it is admissible and for every admissible $S' \subset S$, $S^+ \not\subset S'^+$ holds. Given a DP algorithm for computing admissible sets, we can compute preferred ones by simple subset maximization using our approach. Our approach also allows for computing semi-stable sets, but it is more involved and we omit the details. In Section 4 we give results of experiments on computing admissible, preferred and semi-stable sets.

4 System and Evaluation

We have implemented our approach in a system called D-FLAT^2 by extending the publicly available D-FLAT system [1]. Both systems allow the problem-specific steps of a DP algorithm working on a TD to be specified in ASP. In D-FLAT^2, the user can specify the computation of solution candidates and the system takes care of the second TD traversal that computes counter candidate pointers and performs subset minimization. For the introductory example $\subseteq$-Minimal Sat, it suffices to provide an algorithm encoding that computes models like for Sat and to specify that items corresponding to atoms are optimization items. Similarly, for preferred sets in abstract argumentation, we can start with an encoding for admissible sets and state that all arguments that are in an extension also are optimization items.

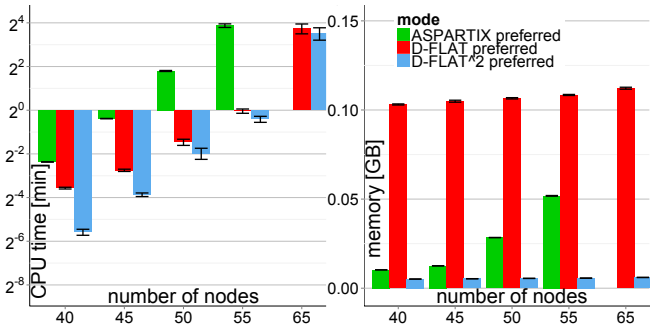

Figure 6 System comparison: Average CPU time (left) and maximum resident set (right).

In this section we focus on experiments for problems from the area of abstract argumentation. We compared systems that implement DP on TDs, namely D-FLAT 1.0.2 and D-FLAT^2 1.0.2. Furthermore, we benchmarked the ASPARTIX system [11] that solves argumentation-related problems directly via ASP. D-FLAT and D-FLAT^2 internally use ASP grounder Gringo 4.4.0 and solver Clasp 3.1.1. The results for ASPARTIX were produced with Gringo 3.0.5, as it is not fully compatible with newer versions of Gringo. For evaluation we used so-called "grid-based" instances, where vertices are arranged on an $n \times m$ matrix and edges connect horizontally, vertically and diagonally neighboring vertices. Each instance was run five times with different TDs, and every run was limited to one hour and three GB of memory.

System comparison. We considered the problem of enumerating all preferred extensions and compared the systems on grid-based instances with 40 to 65 nodes and treewidth 4. Figure 6 illustrates average runtimes and allocated memory together with the 95 % confidence interval. D-FLAT^2 showed the best performance, while D-FLAT is slightly slower and requires more memory. For ASPARTIX we observed timeouts for instances having more than 55 nodes.

Problem comparison. As D-FLAT^2 is based on D-FLAT, we compared these systems on several problems. Moreover, we analyzed the cost of computing preferred and semi-stable sets compared to only obtaining admissible sets. As instances have much more admissible sets than preferred sets, which would bias a performance comparison when doing explicit enumeration, we considered the counting variants of these problems. Results are summarized in Figure 7, where again grid-based instances with treewidth 4 were tested.

When counting admissible sets, D-FLAT^2 requires marginally more time and memory than D-FLAT due to the overhead imposed by using reduced item trees instead of item trees. For preferred sets, the inefficiency of computing redundant counter candidates in D-FLAT becomes evident. On the contrary, in D-FLAT^2 the difference in runtime for counting preferred instead of admissible sets is barely measurable (i.e. within the 95% confidence interval). Here, we observed that subset maximization comes for free for instances of small treewidth. (We also observed this effect in a comparison of Sat with $\subseteq$-Minimal Sat, where the overhead of D-FLAT was even larger.) Finally, for semi-stable sets, D-FLAT was not able solve instances with 500 vertices within the given memory limits. One reason is that for this problem many potential counter candidates have to be computed that turn out to be not even admissible. Thus, our two-phased approach of first computing (not necessarily maximal) solutions and then performing maximization obviously pays off in this case.

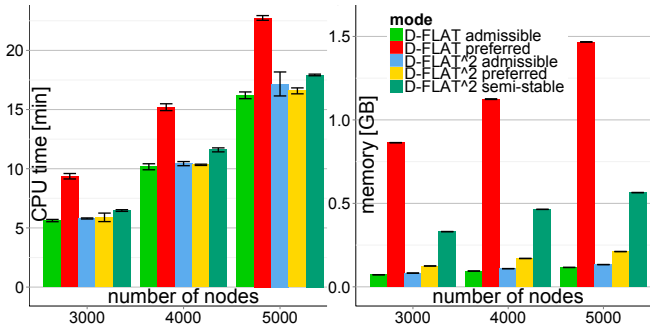

Figure 7 Problem comparison: Average CPU time (left) and maximum resident set (right).

5 Conclusion

In this work we presented a method for solving problems involving subset minimization by means of DP on TDs. Given an algorithm for a version of the problem without minimization, our method allows us to perform the minimization tasks in an automatic and uniform way, thus making the development of such algorithms significantly easier. We have outlined the details of this minimization procedure and shown its effectiveness and efficiency by experimenting with a prototype implementation. Preliminary experiments already indicate that our new approach brings significant advantages in terms of time and memory compared to previous solutions. This makes our method relevant especially for AI problems, as these often require some sort of subset minimization. In the future, we need to test our approach on further problems in order to further improve the new D-FLAT^2 system. Moreover, we plan to extend our approach to problems that are even higher in the polynomial hierarchy than the second level.

References

1 Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proc. JELIA*, volume 8761 of *LNCS*, pages 558–572, 2014.

2 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

3 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

4 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

5 Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

6 Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

7 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

8 Phan M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, 1995.

9 Paul E. Dunne. Computational properties of argument systems satisfying graph-theoretic constraints. *Artif. Intell.*, 171(10-15):701–729, 2007.

10 Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

11 Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.

12 Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP*, 6(1-2):23–60, 2006.

13 Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *TPLP*, 11(4-5):821–839, 2011.

14 Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, 174(1):105–132, 2010.

15 Georg Gottlob, Reinhard Pichler, and Fang Wei. Tractable database design and datalog abduction through bounded treewidth. *Inf. Syst.*, 35(3):278–298, 2010.

16 Georg Gottlob and Stefan Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *Comput. J.*, 51(3):303–325, 2008.

17 Michael Jakl, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Fast counting with bounded treewidth. In *Proc. LPAR*, volume 5330 of *LNCS*, pages 436–450. Springer, 2008.

18 Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI*, pages 816–822, 2009.

19 Ton Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

20 Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

21 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

22 John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artif. Intell.*, 13(1–2):27–39, 1980.

23 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.

24 David Pearce, Hans Tompits, and Stefan Woltran. Characterising equilibrium logic and nested logic programs: Reductions and complexity. *Computing Research Repository*, abs/0906.2228, 2009.

25 Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

26 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.